Building Library Components That Can Use Any MPI Implementation

William Gropp

May 16, 2002

Abstract

The Message Passing Interface (MPI) standard for programming parallel computers is widely used for building both programs and libraries. Two of the strengths of MPI are its support for libraries and the existence of multiple implementations on many platforms. These two strengths are in conflict, however, when an application wants to use libraries built with different MPI implementations. This paper describes several solutions to this problem, based on minor changes to the API. These solutions also suggest design considerations for other standards, particularly those that expect to have multiple implementatations and to be used in concert with other libraries.

The MPI standard [1, 2] has been very successful. There are multiple implementations of MPI for most parallel computers [3], including vendor-optimized versions and several freely-available versions. In addition, MPI provides support for constructing parallel libraries. However, when an application wants to use routines from several different parallel libraries, it must ensure that each library was built with the *same* implementation of MPI. This is due to the contents of the header files '`mpi.h`' (for C and C++) and '`mpif.h`' for Fortran or the MPI module for Fortran 90. Because the goals of MPI included high performance, the MPI standard gives the implementor wide latitude in the specification of many of the datatypes and constants used in an MPI program. For each individual MPI program, this lack of detailed specification of the header files causes no problems; few users are even aware that the specific value of, for example, `MPI_ANY_SOURCE` is not specified by the standard. Only the names are specified.

Because the values of the items defined in the header files are not defined by the standard, software components that use different MPI implementations may not be mixed in a single application. Users currently faced with building an application from multiple libraries must either mandate that a specific MPI implementation be used for all components or that all libraries be built with all MPI implementations. Neither approach is adequate; third-party MPI libraries may only be available for specific MPI implementations and the building and testing each library for each MPI implementation is both time-consuming and difficult to manage; in addition, the risk of picking the wrong version of a library

is high, leading to problems that are difficult to diagnose. Finally, while building each library with each version of MPI is just possible, this solution doesn't scale to other APIs that have multiple implementations. Thus, a solution that allows an application to use *any* implementation of MPI (and, using similar techniques, other libraries) is needed.

There are at least two solutions to this problem. The first is completely generic. It effectively wraps all MPI routines and objects with new versions that work with any MPI implementation by deferring the choice of a specific implementation to link-time (or even runtime if dynamically-linked libraries are used). Fortunately, the emergence of tools for building language-specific interfaces from language-independent descriptions of components can be used; this approach is described in a companion paper [5]. The second is a practical subset for MPI implementations whose opaque objects have the same size (e.g., all four bytes). Neither solution is perfect in the sense that both require some changes to the source code of an existing component that already uses MPI. In many cases, however, these changes could be automated, making their use almost transparent. This paper focuses on the C binding of MPI, with some comments about C++ and Fortran. A brief discussion of the first alternative is followed by a detailed discussion of the second, along with implementation examples.

1 Brief Overview of MPI

The MPI specification itself is language independent. The MPI-1 standard specifies bindings to C and Fortran (then Fortran 77); the MPI-2 standard added bindings for C++ and Fortran 90. MPI programs are required to include the appropriate header file (or module for Fortran 90); for C and C++, this is '`mpi.h`'. The contents of this file include: typedefs for MPI opaque objects, definitions of compile-time constants, definitions of link-time constants (see below), and function prototypes. However, the implementation is given wide latitude in the definition of the definitions of the named objects. For example, the following definitions for the MPI object used to describe data layouts (`MPI_Datatype`) are used by four common implementations:

Implementation	Datatype definition
LAM	`typedef struct _dtype *MPI_Datatype;`
IBM	`typedef int MPI_Datatype;`
SGI	`typedef unsigned int MPI_Datatype;`
MPICH	`typedef int MPI_Datatype;`

While no implementation appears to use a `struct` rather than an `int` or pointer, nothing in the MPI standard precludes doing so.

The MPI standard also requires each routine to be available with both `MPI` and `PMPI` prefixes. For example, both `MPI_Send` and `PMPI_Send` are required. This is called the profiling interface. The intent of the `PMPI` versions is to allow a tool to replace the implementations of the `MPI` routines with code that provides

added functionality (such as collecting performance data); the routine can then use the `PMPI` version to perform the MPI operation. This is a powerful feature that can be exploited in many applications and tools, such as the performance visualization tools Jumpshot [6] and VAMPIR [4].

2 A Generic MPI Component

The most general solution is to create a new MPI component. This can be thought of as a new binding of the MPI standard; it must define every element of MPI. For concreteness, we will call this the GMPI binding, and for each MPI term, this new component will provide a GMPI version. Rather than re-implement MPI, this component is implemented in terms of an MPI implementation, providing for the translation between objects defined by the component and the corresponding object in each MPI implementation. For example, the definition of a generic `MPI_Datatype` might be

```
typedef GMPI_Handle *GMPI_Datatype;
```

The implementation of a simple routine such as `GMPI_Type_free` might look something like

```
int GMPI_Type_free( GMPI_Datatype *type )
{
    MPI_Datatype local_type = GMPI_to_mpi_datatype( type );
    int rc;
    rc = MPI_Type_free( &local_type );
    if (!rc)
        *type = MPI_to_gmpi_datatype( local_type );
    return rc;
}
```

Libraries built with such a component are interoperable; all that is needed for each MPI implementation is an implementation of this new binding in terms of each MPI implementation. However, code must be written to use this new binding of MPI. In addition, because the interface is layered over another MPI implementation, there is added overhead.

Note that because this routine calls the underlying MPI routine and not the PMPI routine, this allows sue of any MPI tool, such as performance visualizers, that exploit the MPI profiling interface. This code also leaves the error handling to the underlying MPI routine; if that error handler returns a value, then this routine will return that same value to the calling environment.

Because this is a complete binding of MPI, all MPI features are available to any code that uses this approach. The lesson for API designers is that at a binding that fully specifies all values enables applications and can be provided along with bindings that trade performance (though lack of detail in the specification) for interoperability.

This solution and the techniques used to implemented it efficiently and quickly are described in more detail in [5].

3 A Generic MPI Header

An alternative solution is to find a common version of the header file that can be used by multiple implementations. This solution is less general than the generic component but has the advantage of requiring fewer changes to existing codes. The approach here is to replace compile-time constants with runtime constants by finding a common data representations for MPI objects. There are some compromises in this approach that are discussed below. However, the advantages are that it requires few changes to existing code and most operations directly call the specific MPI routines (with no overhead).

The approach is as follows: define a new header file that replaces most compile-time values with runtime values. These values are then set when the initialization of MPI takes place, either though a version `MPI_Init` or a special `GMPI_Init`. This initialization is described below. To make it clear that a special initialization routine is required, we will refer to `GMPI_Init` in the text below.

The header file '`mpi.h`' contains the following items that must be handled:

Compile-time values. These include the constants defining error classes (e.g., `MPI_ERR_TRUNCATE`) and special values for parameters (e.g., `MPI_ANY_SOURCE`). These can be replaced with runtime values that are initialized by `GMPI_Init`.

A special case are the NULL objects such as `MPI_COMM_NULL`. Most implementations define all NULL objects the same way (either as zero or -1). However, the MPICH-2 implementation encodes the object type in each object handle, including the null object handles. Thus, these terms must cannot be defined at compile-time.

Compile-time values used in declarations. This includes the constants defining maximum string sizes such as `MPI_MAX_ERROR_STRING`. For many cases, these can be replace by either the maximum or the minimum over the supported implementations. For example,

Implementation	Size of `MPI_MAX_ERROR_STRING`
SGI	256
IBM	128
LAM	256
MPICH	512

Defining `MPI_MAX_ERROR_STRING` as 512 is adequate for all of these MPI-1 implementations, since this value is used only to declare strings that are used as output parameters. Other values are input, such as `MPI_MAX_INFO_KEY`. In this case, the minimum value should be used. While this might seem like a limitation, a truely portable code would need to limit itself to the minimum value used in an supported MPI implementation in any event, so this is not a limitation in practice.

Link-time constants. MPI defines a number of items that are link-time, rather than compile-time, constants. The predefined MPI datatypes such as

MPI_INT belong to this category. For most users, the difference isn't apparent; it only comes up when users try to use one of these in a place where compile-time constants are required by the language (such as case labels in a switch statement). These are actually easier to implement than the compile-time constants because it is correct to define them as values that are initialized at run time.

Opaque objects. These are both the most difficult and the easiest. In practice, most implementations define these as objects of a particular size. They may be `int`s or pointers, but never anything else. On many platforms, `int`s and pointers are the same size. For such platforms, we can simply pick one form (such as `int`) and use that.

If the size of these objects is different among different implementations, then there is no easy solution. We cannot pick the largest size because some MPI operations use arrays of opaque objects (e.g., `MPI_Waitsome` or `MPI_Type_struct`).

One additional compilication is the handle conversion functions introduced in MPI-2. These convert between the C and Fortran representations of the handles. For many implementations, these are simply cast operations. However, for greatest flexibility, GMPI versions of these should be used, e.g., `GMPI_Type_f2c` instead of `MPI_Type_f2c`. This defers until link time the definition of these functions.

Defined objects (status). The most difficult case is `MPI_Status`. MPI defines this as a `struct` with some defined members such as `MPI_TAG`. An implementation is allowed to add its own members to this structure. Because of this, the size of `MPI_Status` and the layout of the members may be (and is) different in each implementation. The only solution to this is to provide routines to allocate and free `MPI_Status` objects and to provide separate routines to access all of the elements. For example, instead of

```
MPI_Status status;
...
MPI_Recv( ..., &status );
if (status.MPI_TAG == 10 || status.MPI_SOURCE == 3) ...
```

you must use special routines such as

```
MPI_Status *status_p = GMPI_Status_create(1);
MPI_Recv( ..., status_p );
if (GMPI_Status_get_tag( status_p ) == 10 ||
    GMPI_Status_get_source( status_p )) ...
GMPI_Status_free( status_p, 1 );
```

Fortunately, many MPI programs don't need or use `status`. These programs should use the MPI-2 values `MPI_STATUS_NULL` or `MPI_STATUSES_NULL` (which, of course, are initialized at run time).

Defined pointers. MPI also defines a few constant pointers, including `MPI_BOTTOM` and `MPI_STATUS_NULL`. For many implementations, these are just `(void *)0`. Like the most of the other constants, these can be set at initialization time.

`MPI_STATUS_NULL` is a special case. This is not part of MPI-1 but was added to MPI-2. If an MPI implementation does not define it, a dummy `MPI_Status` may be used instead. This will not work for the corresponding `MPI_STATUSES_NULL` which is used for arguments that require an array of `MPI_Status`, but is sufficient for most uses.

The complete list of functions that must be used for accessing information in `MPI_Status` is:

```
create and free one status
get values for tag, source, error for one status
create and free an array of status
get values for tag, source, error for an element of an array of status
```

3.1 Using the generic header

Applications are compiled in the same way as other MPI programs, using the '`gmpi.h`' header file instead of '`mpi.h`'. Linking is almost the same, except one additional library is needed: the implementation of the GMPI routines in terms of a particular MPI implementation. For example, consider an application that uses GMPI and a library component that is also built with GMPI. The link line looks something like

```
# Independent of MPI implementation (generic mpi.h in /usr/local/gmpi)
cc -c myprog.c -I/usr/local/gmpi
cc -c mylib.c -I/usr/local/gmpi
ar cr libmylib.a mylib.o
ranlib libmylib.a
# For MPICH
cc -o myprog myprog.o -lmylib -lgmpitompich -L/usr/local/mpich -lmpich
# For LAM/MPI
cc -o myprog myprog.o -lmylib -lgmpitolam -L/usr/local/lam -lmpi
```

With this approach, only one compiled version of each library or object file is required. In addition, for each MPI implementation, a single library implementing GMPI in terms of that implementation is required.

To illustrate this, we show a simple library that can be used, in compiled form, with two different MPI implementations on a Beowulf cluster. The simple library provides two versions of a numerical inner product; one which uses `MPI_Allreduce` and is fast and one which preserves evaluation order for the operation (unlike real numbers, floating-point arithmetic is not associative, so the order of evaluation can affect the final result). The code for the library routine is

```
#include "mpi.h"
```

```
double parallel_dot( const double restrict u[],
                     const double restrict v[], int n, int ordered,
                     MPI_Comm comm )
{
    int    rank, size, i;
    double temp, result;

    if (ordered) {
        MPI_Comm tempcomm;
        /* A sophisticated version would cache the duplicated communicator */
        MPI_Comm_dup( comm, &tempcomm );
        MPI_Comm_rank( tempcomm, &rank );
        MPI_Comm_size( tempcomm, &size );
        if (rank != 0) {
            MPI_Recv( &temp, 1, MPI_DOUBLE, rank-1, 0, tempcomm,
                      MPI_STATUS_NULL );
        }
        else {
            temp = 0.0;
        }
        for (i=0; i<n; i++) {
            temp += u[i] * v[i];
        }
        if (rank != size-1) {
            MPI_Send( &temp, 1, MPI_DOUBLE, rank+1, 0, tempcomm );
        }
        MPI_Bcast( &temp, 1, MPI_DOUBLE, size-1, tempcomm );
        MPI_Comm_free( &tempcomm );
        result = temp;
    }
    else {
        temp = 0.0;
        for (i=0; i<n; i++) {
            temp += u[i] * v[i];
        }
        MPI_Allreduce( &temp, &result, 1, MPI_DOUBLE, MPI_SUM, comm );
    }
    return result;
}
```

Note that this code uses the communicator that is passed into the routine. The following shell commands show how easy it is to build this library so that it may be used by either MPICH or LAM/MPI. The main program is in `myprog` and includes the call to `MPI_Init` and `MPI_Finalize`.

```
% cc -c -I/usr/local/gmpi/include dot.c
% ar cr libmydot.a dot.o
```

```
% ranlib libmydot.a
% cc -c -I/usr/local/gmpi/include myprog.c
% /usr/local/mpich/bin/mpicc -o myprog myprog.o -lmydot -lgmpitompich
% /usr/local/mpich/bin/mpirun -np 4 myprog
% /usr/local/lammpi/bin/mpicc -o myproc myprog.o -lmydot -lgmpitolam
% /usr/local/lammpi/bin/mpirun -np 4 myprog
```

4 Generic MPI for C++ and Fortran

What goes here? C++ uses can use a new class or versions of the C++ to C wrappers that defer most choices to link time; because the C++ binding uses methods for construction and access, even for status, C++ libraries can use a generic MPI header with greater ease.

Fortran needs only replace `parameter` statements with values in `common`. However, status becomes a problem. For individual status, we can just take a maximum. For arrays, that's not possible, unless we provide routines to access the array elements as well.

5 Remarks for API Developers

(Sumarize the lessons: access methods for any object with variable size. Try to minimize compile-time constants; define appropriate min/max values so that programs can be portable; arrays of structures with undefined sizes are bad.)

Tradeoffs. Briefly discuss why MPI may the decisions that it did.

A Generic mpi.h header file

```
/*
 *  $Id$
 *
 *  (C) 2002 by Argonne National Laboratory.
 *      All rights reserved.  See COPYRIGHT in top-level directory.
 */

#ifndef _GMPI_INCLUDE
#define _GMPI_INCLUDE

/* 3. Opaque Objects */
typedef int MPI_Comm;
typedef int MPI_Group;
typedef int MPI_Op;
typedef int MPI_Datatype;
typedef int MPI_Errhandler;
typedef int MPI_Request;
typedef int MPI_Info;
```

```
/* 1. Run-time values */

/* group and communicator comparison */
extern int MPI_IDENT;
extern int MPI_CONGRUENT;
extern int MPI_SIMILAR;
extern int MPI_UNEQUAL;

/* Datatypes */
extern MPI_Datatype MPI_CHAR;
extern MPI_Datatype MPI_UNSIGNED_CHAR;
extern MPI_Datatype MPI_BYTE;
extern MPI_Datatype MPI_SHORT;
extern MPI_Datatype MPI_UNSIGNED_SHORT;
extern MPI_Datatype MPI_INT;
extern MPI_Datatype MPI_UNSIGNED;
extern MPI_Datatype MPI_LONG;
extern MPI_Datatype MPI_UNSIGNED_LONG;
extern MPI_Datatype MPI_FLOAT;
extern MPI_Datatype MPI_DOUBLE;
extern MPI_Datatype MPI_LONG_DOUBLE;
extern MPI_Datatype MPI_LONG_LONG_INT;
extern MPI_Datatype MPI_LONG_LONG;
extern MPI_Datatype MPI_PACKED;
extern MPI_Datatype MPI_LB;
extern MPI_Datatype MPI_UB;
extern MPI_Datatype MPI_FLOAT_INT;
extern MPI_Datatype MPI_DOUBLE_INT;
extern MPI_Datatype MPI_LONG_INT;
extern MPI_Datatype MPI_SHORT_INT;
extern MPI_Datatype MPI_2INT;
extern MPI_Datatype MPI_LONG_DOUBLE_INT;

/* Fortran types */
extern MPI_Datatype MPI_COMPLEX;
extern MPI_Datatype MPI_DOUBLE_COMPLEX;
extern MPI_Datatype MPI_LOGICAL;
extern MPI_Datatype MPI_REAL;
extern MPI_Datatype MPI_DOUBLE_PRECISION;
extern MPI_Datatype MPI_INTEGER;
extern MPI_Datatype MPI_2INTEGER;
extern MPI_Datatype MPI_2COMPLEX;
extern MPI_Datatype MPI_2DOUBLE_COMPLEX;
extern MPI_Datatype MPI_2REAL;
extern MPI_Datatype MPI_2DOUBLE_PRECISION;
extern MPI_Datatype MPI_CHARACTER;

/* Communicators */
extern MPI_Comm MPI_COMM_WORLD;
extern MPI_Comm MPI_COMM_SELF;
```

```
/* Groups */
extern MPI_Group MPI_GROUP_EMPTY;

/* Collective operations */
extern MPI_Op MPI_MAX;
extern MPI_Op MPI_MIN;
extern MPI_Op MPI_SUM;
extern MPI_Op MPI_PROD;
extern MPI_Op MPI_LAND;
extern MPI_Op MPI_BAND;
extern MPI_Op MPI_LOR;
extern MPI_Op MPI_BOR;
extern MPI_Op MPI_LXOR;
extern MPI_Op MPI_BXOR;
extern MPI_Op MPI_MINLOC;
extern MPI_Op MPI_MAXLOC;

/* Permanent key values */
/* C Versions (return pointer to value) */
extern int MPI_TAG_UB;
extern int MPI_HOST;
extern int MPI_IO;
extern int MPI_WTIME_IS_GLOBAL;

/* Define some null objects */
extern MPI_Comm       MPI_COMM_NULL;
extern MPI_Op         MPI_OP_NULL;
extern MPI_Group      MPI_GROUP_NULL;
extern MPI_Datatype   MPI_DATATYPE_NULL;
extern MPI_Request    MPI_REQUEST_NULL;
extern MPI_Errhandler MPI_ERRHANDLER_NULL;

/* 2. Compile-time values */
/* Output values (max over implementations) */
#define MPI_MAX_PORT_NAME 256
#define MPI_MAX_PROCESSOR_NAME 256
#define MPI_MAX_ERROR_STRING   512
#define MPI_MAX_NAME_STRING     63

/* Input or input/output values (min over implementations) */
# define MPI_MAX_INFO_KEY       255
# define MPI_MAX_INFO_VAL      1024

/* Pre-defined constants */
extern int MPI_UNDEFINED;
extern int MPI_UNDEFINED_RANK;
extern int MPI_KEYVAL_INVALID;

/* Upper bound on the overhead in bsend for each message buffer */
extern int MPI_BSEND_OVERHEAD;
```

```
/* Topology types */
extern int MPI_GRAPH;
extern int MPI_CART;

#define MPI_BOTTOM      (void *)0

extern int MPI_PROC_NULL;
extern int MPI_ANY_SOURCE;
extern int MPI_ROOT;
extern int MPI_ANY_TAG;

/* MPI Error handlers.  */
typedef void (MPI_Handler_function) ( MPI_Comm *, int *, ... );

extern MPI_Errhandler MPI_ERRORS_ARE_FATAL;
extern MPI_Errhandler MPI_ERRORS_RETURN;

/* Make the C names for the null functions all upper-case.  Note that
   this is required for systems that use all uppercase names for Fortran
   externals.  */
/* MPI 1 names */
/* ??? These need wrappers? ??? */
#define MPI_NULL_COPY_FN   MPIR_null_copy_fn
#define MPI_NULL_DELETE_FN MPIR_null_delete_fn
#define MPI_DUP_FN         MPIR_dup_fn

/* MPI 2 names */
#define MPI_COMM_NULL_COPY_FN MPI_NULL_COPY_FN
#define MPI_COMM_NULL_DELETE_FN MPI_NULL_DELETE_FN
#define MPI_COMM_DUP_FN MPI_DUP_FN

/* User combination function */
typedef void (MPI_User_function) ( void *, void *, int *, MPI_Datatype * );

/* MPI Attribute copy and delete functions */
typedef int (MPI_Copy_function) ( MPI_Comm, int, void *, void *, void *, int * );
typedef int (MPI_Delete_function) ( MPI_Comm, int, void *, void * );

/* These constants can be left constant, since all implementations must have
   the same values for them.  In fact, they are required to be compile-time
   values. */
#define MPI_VERSION    1
#define MPI_SUBVERSION 2
#define MPICH_NAME     1

/* for the datatype decoders */
extern int MPI_COMBINER_NAMED;
extern int MPI_COMBINER_CONTIGUOUS;
extern int MPI_COMBINER_VECTOR;
extern int MPI_COMBINER_HVECTOR;
```

```
extern int MPI_COMBINER_INDEXED;
extern int MPI_COMBINER_HINDEXED;
extern int MPI_COMBINER_STRUCT;

/* for info */
extern int MPI_INFO_NULL;

/* for subarray and darray constructors */
extern int MPI_ORDER_C;
extern int MPI_ORDER_FORTRAN;
extern int MPI_DISTRIBUTE_BLOCK;
extern int MPI_DISTRIBUTE_CYCLIC;
extern int MPI_DISTRIBUTE_NONE;
extern int MPI_DISTRIBUTE_DFLT_DARG;

typedef long MPI_Aint;
typedef int MPI_Fint;
/* What do we want to do about status ? */
typedef struct { int dummy[10]; } MPI_Status;
extern MPI_Status *MPI_STATUS_NULL;

/* Handle conversion types/functions */

/* Programs that need to convert types used in MPICH should use these */
/* These need to be fixed.  For some impl sets, they can remain; for others,
   they'll need to be wrappers. Note that these are routines for lam.
   The thing to do is to provide these routines for the implementations
   that do not provide them */
#define MPI_Comm_c2f(comm) (MPI_Fint)(comm)
#define MPI_Comm_f2c(comm) (MPI_Comm)(comm)
#define MPI_Type_c2f(datatype) (MPI_Fint)(datatype)
#define MPI_Type_f2c(datatype) (MPI_Datatype)(datatype)
#define MPI_Group_c2f(group) (MPI_Fint)(group)
#define MPI_Group_f2c(group) (MPI_Group)(group)
/* MPI_Request_c2f is a routine in src/misc2 */
#define MPI_Request_f2c(request) (MPI_Request)MPIR_ToPointer(request)
#define MPI_Op_c2f(op) (MPI_Fint)(op)
#define MPI_Op_f2c(op) (MPI_Op)(op)
#define MPI_Errhandler_c2f(errhandler) (MPI_Fint)(errhandler)
#define MPI_Errhandler_f2c(errhandler) (MPI_Errhandler)(errhandler)

/* For new MPI-2 types */
#define MPI_Win_c2f(win)   (MPI_Fint)(win)
#define MPI_Win_f2c(win)   (MPI_Win)(win)

#define MPI_STATUS_IGNORE (MPI_Status *)0
#define MPI_STATUSES_IGNORE (MPI_Status *)0

/* For supported thread levels */
extern int MPI_THREAD_SINGLE;
```

```
extern int MPI_THREAD_FUNNELED;
extern int MPI_THREAD_SERIALIZED;
extern int MPI_THREAD_MULTIPLE;

/* MPI's error classes */
#define MPI_SUCCESS          0
extern int MPI_ERR_BUFFER;
extern int MPI_ERR_COUNT;
extern int MPI_ERR_TYPE;
extern int MPI_ERR_TAG;
extern int MPI_ERR_COMM;
extern int MPI_ERR_RANK;
extern int MPI_ERR_ROOT;
extern int MPI_ERR_TRUNCATE;
extern int MPI_ERR_GROUP;
extern int MPI_ERR_OP;
extern int MPI_ERR_REQUEST;
extern int MPI_ERR_TOPOLOGY;
extern int MPI_ERR_DIMS;
extern int MPI_ERR_ARG;
extern int MPI_ERR_OTHER;
extern int MPI_ERR_UNKNOWN;
extern int MPI_ERR_INTERN;
extern int MPI_ERR_IN_STATUS;
extern int MPI_ERR_PENDING;

/* New MPI-2 Error classes */
extern int MPI_ERR_FILE;
extern int MPI_ERR_ACCESS;
extern int MPI_ERR_AMODE;
extern int MPI_ERR_BAD_FILE;
extern int MPI_ERR_FILE_EXISTS;
extern int MPI_ERR_FILE_IN_USE;
extern int MPI_ERR_NO_SPACE;
extern int MPI_ERR_NO_SUCH_FILE;
extern int MPI_ERR_IO;
extern int MPI_ERR_READ_ONLY;
extern int MPI_ERR_CONVERSION;
extern int MPI_ERR_DUP_DATAREP;
extern int MPI_ERR_UNSUPPORTED_DATAREP;

/* MPI_ERR_INFO is NOT defined in the MPI-2 standard.  I believe that
   this is an oversight */
extern int MPI_ERR_INFO;
extern int MPI_ERR_INFO_KEY;
extern int MPI_ERR_INFO_VALUE;
extern int MPI_ERR_INFO_NOKEY;

extern int MPI_ERR_NAME;
extern int MPI_ERR_NOMEM;
```

```
extern int MPI_ERR_NOT_SAME;
extern int MPI_ERR_PORT;
extern int MPI_ERR_QUOTA;
extern int MPI_ERR_SERVICE;
extern int MPI_ERR_SPAWN;
extern int MPI_ERR_UNSUPPORTED_OPERATION;
extern int MPI_ERR_WIN;

extern int MPI_ERR_LASTCODE;

/* End of MPI's error classes */

/* Bindings are the same as for MPI and are omitted here */
#include "gmpibinding.h"

/* Add the *new* routines */


#endif
```

B Implementing the generic header file and interface library

The first step in the implementation is the creation of the '`gmpi.h`' file from the '`mpi.h`' file provided by a particular implementation. For the MPI implementations that we have considered, this can be handled with a simple `perl` program that searches the include file for `#define` or `enum` definitions of the various MPI constants. In addition, the typedefs are extracted; this allows us to ensure that the typedefs have compatible sizes.

In addition, the necessary routines are provided by a '`gmpitoimpl.c`' file; this contains an implementation of the routines to manage `MPI_Status` elements. It also includes the implementation specific header file created above; this provides a definition of most of the MPI terms. The complete implementation is

```
/*
   For each implementation, use the perl script creategmpi to create the
   file gmpiimpl.h, then use -I to point the compiler at the correct version
   (e.g., put each into a separate directory).
*/

/* This header file defines all of the MPI objects with link-time values.
   This must also contain the TRUE definition of MPI_Status */
#include "gmpiimpl.h"

#include <unistd.h>
```

```
/* Routines to create, access, and destroy status objects.
   This approach uses the smallest number of routines, in exchange for
   making any status routine require a count or an index argument */
MPI_Status *GMPI_Status_create( int n )
{
    MPI_Status *p;

    p = (MPI_Status *)malloc( n * sizeof(MPI_Status) );
    if (!p) MPI_Abort( MPI_COMM_WORLD, 1 );

    return p;
}

void GMPI_Status_free( MPI_Status *p )
{
    free( p );
}

int GMPI_Status_get_tag( MPI_Status *p, int idx )
{
    return p[idx].MPI_TAG;
}

int GMPI_Status_get_source( MPI_Status *p, int idx )
{
    return p[idx].MPI_SOURCE;
}

int GMPI_Status_get_error( MPI_Status *p, int idx )
{
    return p[idx].MPI_ERROR;
}

int GMPI_Status_get_count( MPI_Status *p, int idx )
{
    int count;
    MPI_Get_count( &p[idx], &count );
    return count;
}

#ifdef GMPI_NEEDS_CONVERTERS
/* Use this section for MPI implementations that do *not* use functions
   for the f2c and c2f routines.  If necessary, break these into separate
   sections as required by the supported implementations. */
#endif
```

References

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[2] Message Passing Interface Forum. MPI2: A Message Passing Interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–299, 1998.

[3] MPI implementations. `http://www.mcs.anl.gov/mpi/implementations.html`.

[4] Vampir 2.0 – Visualization and Analysis of MPI Programs. `http://www.pallas.de/pages/vampir.htm`.

[5] Barry Smith. Generic MPI with BABL or something like that. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 2002.

[6] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.